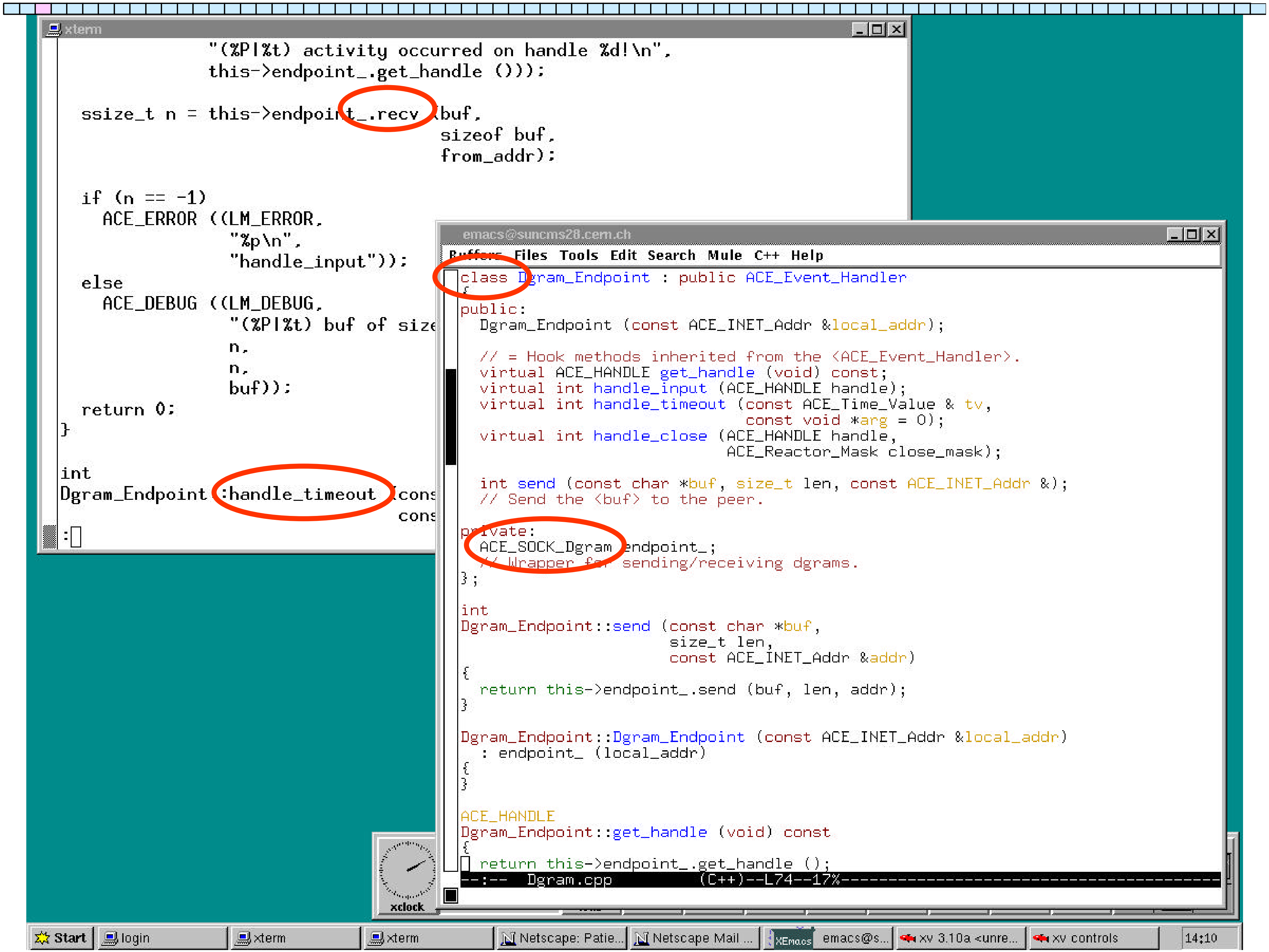
ACE™ - The ADAPTIVE Communication Environment

Johannes Gutleber
Vienna University of Technology, Austria
CERN, Switzerland

An
Adaptive Communication Environment


xterm
"(%P|%t) activity occurred on handle %d!\n",
this->endpoint_.get_handle ());
ssize_t n = this->endpoint_.recv (buf,
sizeof buf,
from_addr);
if (n == -1)
ACE_ERROR ((LM_ERROR,
"%p\n",
"handle_input"));
else
ACE_DEBUG ((LM_DEBUG,
"(%P|%t) buf of size
n,
n,
buf));
return 0;
}
int
Dgram_Endpoint::handle_timeout (cons
cons
emacs@suncms28.cern.ch
Buffers Files Tools Edit Search Mule C++ Help
class Dgram_Endpoint : public ACE_Event_Handler
{
public:
Dgram_Endpoint (const ACE_INET_Addr &local_addr);
// = Hook methods inherited from the <ACE_Event_Handler>.
virtual ACE_HANDLE get_handle (void) const;
virtual int handle_input (ACE_HANDLE handle);
virtual int handle_timeout (const ACE_Time_Value & tv,
const void *arg = 0);
virtual int handle_close (ACE_HANDLE handle,
ACE_Reactor_Mask close_mask);
int send (const char *buf, size_t len, const ACE_INET_Addr &);
// Send the <buf> to the peer.
private:
ACE_SOCK_Dgram endpoint_;
// Wrapper for sending/receiving dgrams.
};
int
Dgram_Endpoint::send (const char *buf,
size_t len,
const ACE_INET_Addr &addr)
{
return this->endpoint_.send (buf, len, addr);
}
Dgram_Endpoint::Dgram_Endpoint (const ACE_INET_Addr &local_addr)
: endpoint_ (local_addr)
{
}
ACE_HANDLE
Dgram_Endpoint::get_handle (void) const
{
return this->endpoint_.get_handle ();
--:-- Dgram.cpp (C++)--L74--17%----
xclock
Start
login
xterm
xterm
Netscape: Patie...
Netscape Mail ...
XEmacs
emacs@s...
xv 3.10a <unre...
xv controls
14:10

Overview

- Object Oriented Terminology
- ACE Wrappers
- Streams
- Message Demultiplexing

45 minutes

- *Break!*

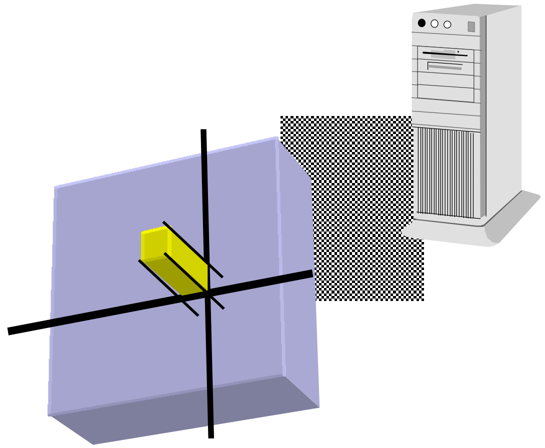
20 minutes

- Service Configuration
- Tasks and Active Objects
- Testimonies

45 minutes

Example

- xy-table, detector test bed
- Radioactive source moves over area
- At each position take data
- Analyse data and store to disk



A Traditional Approach

- Write C Program that contains
  - Control of table
  - Analysis of data
  - Transfer to disk
- Problems that could occur (test beams?)
  - CPU power to small for all tasks -> ***distribute***
  - Local disk space not big enough -> ***transfer data***
  - Include different detectors & readouts -> ***configure***

Modification may be difficult, due to lack of abstraction!

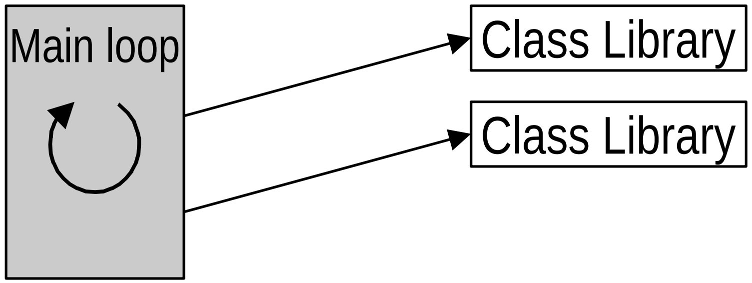
Toolkit/Framework Approach

- Control/analysis tasks are ***encapsulated***
  - Submit required version to scheduler (AO)
  - If CPU power insufficient → execute on other CPU
- Communication is encapsulated
  - Easier to add/change disk and network access
- Tools alleviate from ***synchronisation*** issues
  - … that one tempts to forget anyway ...

This approach does not solve performance problems!

Toolkit

- A ***library*** of ***related classes***.
- The OO equivalent of subroutine libraries.
  - General purpose lists, the C++ IO stream, Mutexes
- The programmer writes the main body of the application and calls the code he wants to use.

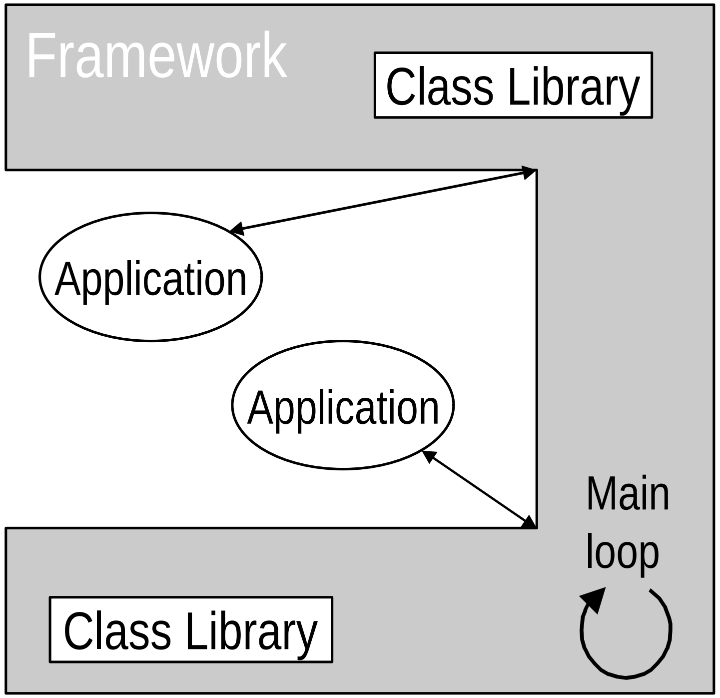


Classes and Components

- **Classes** in an object-oriented toolkit
  - Represent the *useable entities*
  - A class corresponds to a resource
  - Functions for *operating on a resource* are provided
  - e.g. IPC (file descriptor - open, close, read, write)
- **Components** in a toolkit
  - Are *collaborating classes*
  - *A functionality* is presented through a *clean interface*
  - e.g. IPC streams for UDP, TCP, VME: a >> b

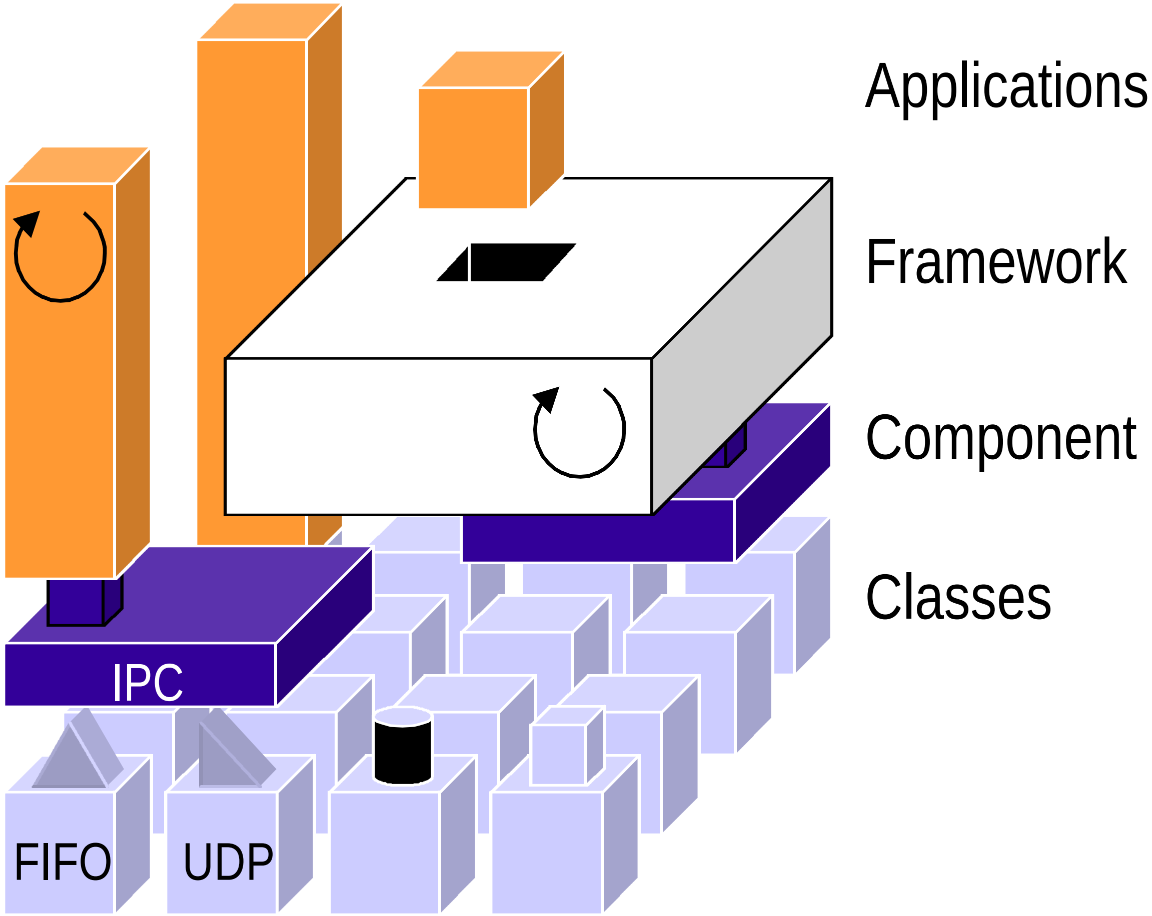
Framework

- A set of *cooperating* classes that make up a *reusable design for* a *specific* class of *software*.
- It defines ***how*** objects ***collaborate***, their responsibilities and the ***thread of control***.
- The programmer reuses the main body and writes the code it calls.



Toolkit Relation Overview

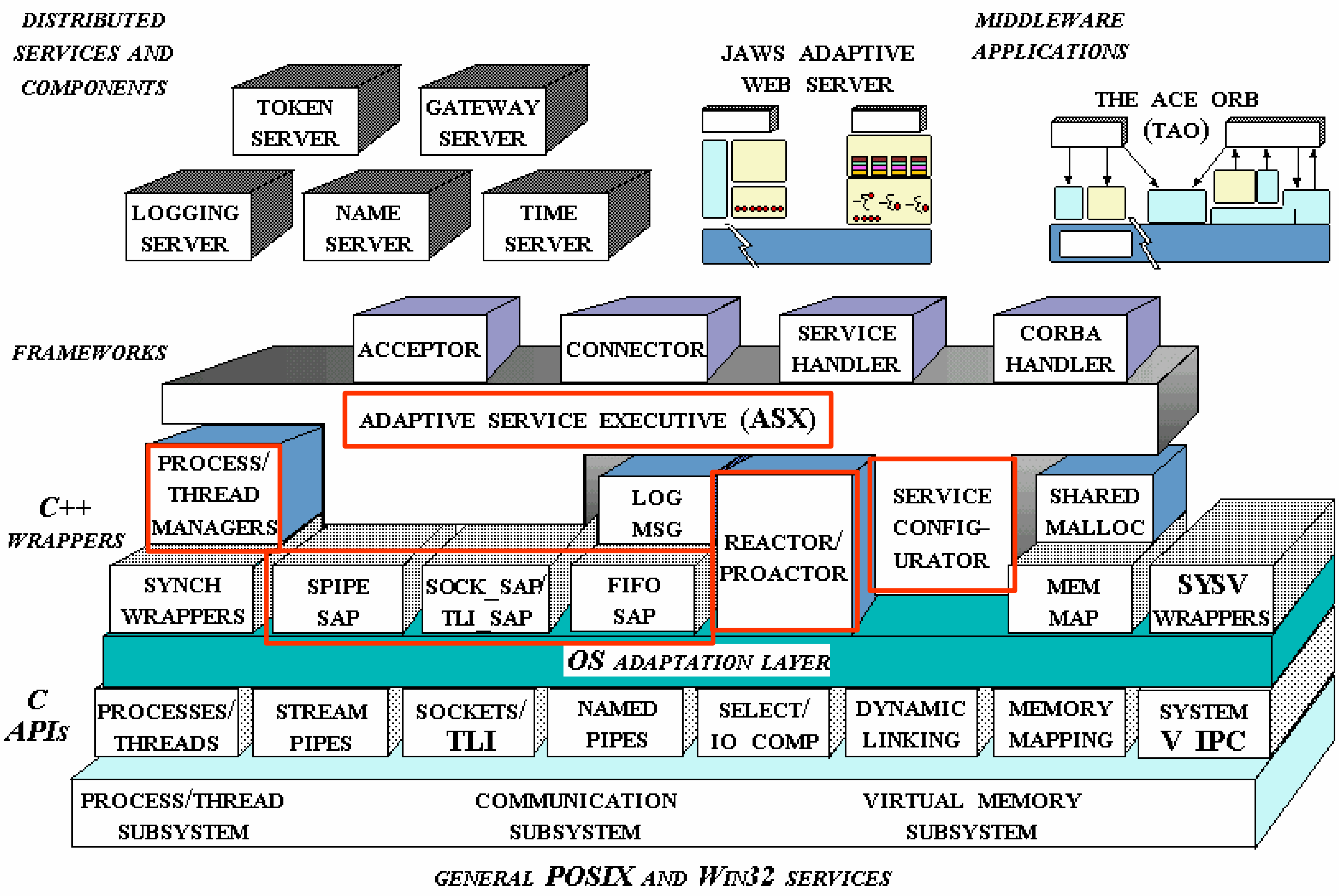
Frameworks and Components have standardised interfaces, although their implementations may be different for different cases. Both consist of collaborating classes.



ACE

- ACE is a multi-layer object-oriented ***toolkit*.**
  - It ***comprises frameworks*** and ***components***.
- Implements several ***Design Patterns.***
  - A pattern describes a solution for a problem that occurs over and over again in a general way.
- Aims at achieving platform independence.
- Can be used to
  - implement applications or
  - framework extensions.

Components of ACE



OS Adaption Layer

- C++ **wrappers** shield upper levels from different operating system APIs (Posix, VxWorks, Win32).
- Provides access to different thread and synchronization packages.
- Eases access to different IPC mechanisms.
- Provides integration of OS calls into C++ code.

It facilitates portability,
it does not provide a Virtual Machine!

Object-Oriented OS API

Although modern OS provide similar functionality, the interfaces are different.

API	Win32	UNIX	VxWorks
SemaphoreID	String	Number	Number
Scheduler	policy	priority+policy	priority
New process	CreateProcess	fork/exec	fork/exec
File Read	ReadFile/overlap	pread	lseek/read
Thread create	*AfxBeginThread*	*pthread_create*	*taskSpawn*
Main	argc/argv/env	argc/ergv/env	spa main args

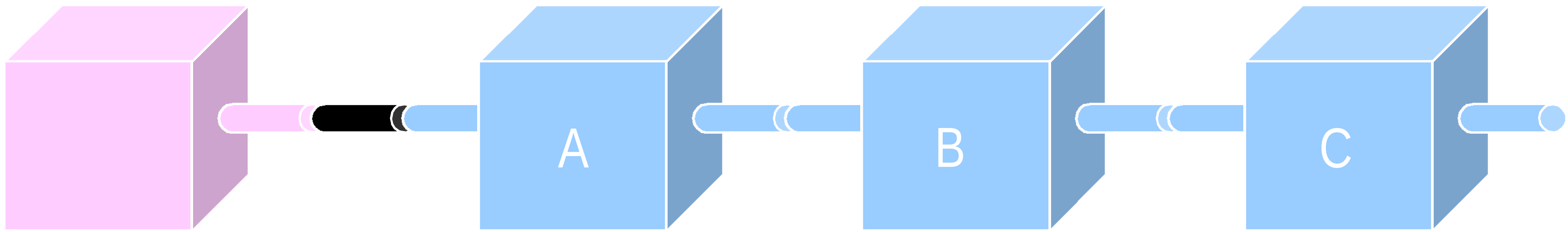
The ACE_OS:: Namespace

- A name space with **one operating system API** for all supported platforms (*best effort only*).
- Input/Output facilities to work with handles
- **Handles** that can be used with any IPC form.

  IPC SAP provide common operations and address classes for pipes, queues, sockets, streams.
- Threads, processes, locks and signals.
- Functions that may not be supported everywhere
  - e.g.: thread suspend/resume (**OS.h file**)

Adaptive Service eXecutive

- A Framework for connecting services in order to build a new application.
- Pipes to connect programs: 1964 McIlroy
- Components with narrow interface: 1969 McIlroy
- Streams as pattern: 1976 Dave Parnas
- Support software toolkit: 1994 D. Schmidt



Ingredients

- Uniform interface to all ***message oriented IPC*** mechanisms (the IPC SAP):
  - `open, close, send, recv, send_n, recv_n`
  - Allows easy reconfiguration of communication software (exchange of transport layer).
- Offers a ***Processing Stream*** facility (Threads).
- A ***service*** may be ***exchange***d at run-time.
- ***Event Processor*** components.
  - (`Acceptor`, `Svc_Handler`, `Event_Handler`).

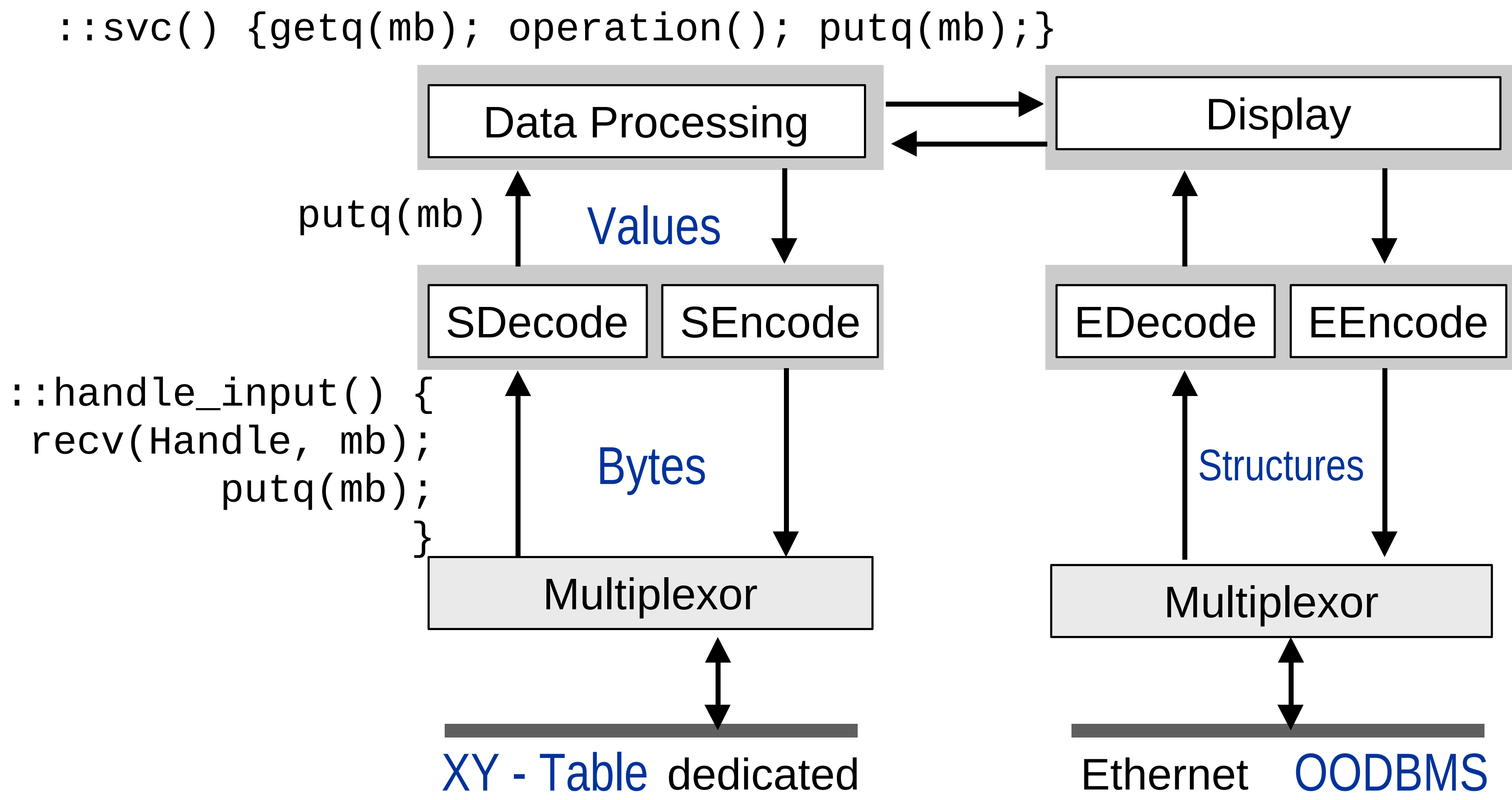
The Stream Facility

- Uses inheritance and object composition to link together service Modules.
  - Inherit from a Thread class and provide service
  - compose them with IPC links
- A ***Stream*** is an object to *configure* and *execute* *services*. It *consists of* inter-connected ***Modules***.
- *Modules* are objects that *decompose* the *application* into a series of interconnected layers. They are the stream chain elements.

The ASX Stream Class

- Example: *xy-table DAQ station*
  - receive x,y values from serial line
  - calculate dx/dt, dy/dt and format the values
  - send them to operator over ethernet
  - same thing in other direction for control

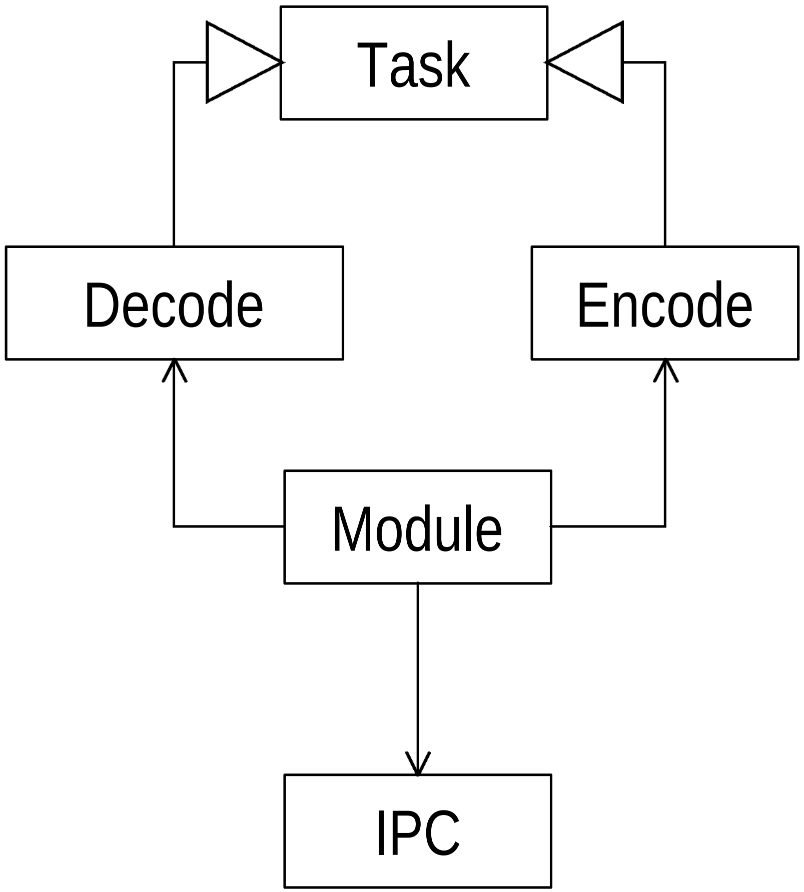
Design with Streams



Important Participants

- `Push, pop` (Stream class)
  - Add/remove a modules to/from the ***stream***.
- `put/get` (Task class)
  - Insert/remove message to/from a stream ***queue***.
- `svc`
  - Service routine of a *Service Handler* class or a ***Task*** class. Within this thread of control data can be received, processed and forwarded.

Make it work




```
Task* Decode, Encode;
Task* DataTrans;
Module A("Serial",
        Decode, Encode);
Module B("DtProcess",
        DtProcess, DtProcess);
Module C("ENet",
        Encode, Decode);
[…]
Stream mainStream;
mainStream.push(A);
mainStream.push(B);
mainStream.push(C);
[…]

mainStream.wait();
```

Open SDecode

```
class SDecode: public Task<ACE_SYNCH> {
  public:
    virtual int svc();
    virtual int open();
    virtual int put(…
  private:
    ACE_TTY_IO        dev;
    ACE_DEV_Connector con;
};

int SDecode::open()
{ // read from serial line and pass to analyzer
  con.connect (dev, ACE_DEV_ADDR ("/dev/somedevice");
  ACE_TTY_IO::Serial_Params params;
  params.baudrate = 9600;
  […]
  dev.control (ACE_TTY_IO::STEPARAMS, &params);}
```

SDecode

```
int SDecode::svc()
{ // read from serial line and pass to analyzer
  while (end != 1) {
    dev.recv_n(&readBuffer, sizeof(readBuffer));
    XYTData xytdata(&readBuffer);
    mb = new ACE_Message_Block(xytdata);
    this->put_next(mb); // async if next has svc
  }
}

int SDecode::put(ACE_Message_Block* m,
                 ACE_Time_Value* timeout)
{ // SDecode never gets anything from other tasks
  // aparts from the message to stop
  end = 1;
  this->release(mb);
}
```

SEncode

```
int SEncode::svc()
{ // read from in queue and drive x-y table
  while (1) {
    this->getq(mb);
    if (mb->msg_type() != ACE_Message_Block::MB_HANGUP)
      break;
    dev.send_n(… // Send data to x-y table
    this->release(mb)
  }
  this->sibling->put(mb); // pass to other task in module
  return 0;
}

int SEncode::put(ACE_Message_Block *m, ACE_Time_Value *to)
{ // Called by other threads.
  // Just enqueue message into local queue
  this->putq(mb);
}
```

Variation I of the Theme

- Have only one class instead of Encode/Decode
  - share the device (explicit synchronisation)
  - in `svc` routine, alternatively perform tasks
    - `if (this->is_reader()) read_device`
    - `if (this->is_writer()) write_device`

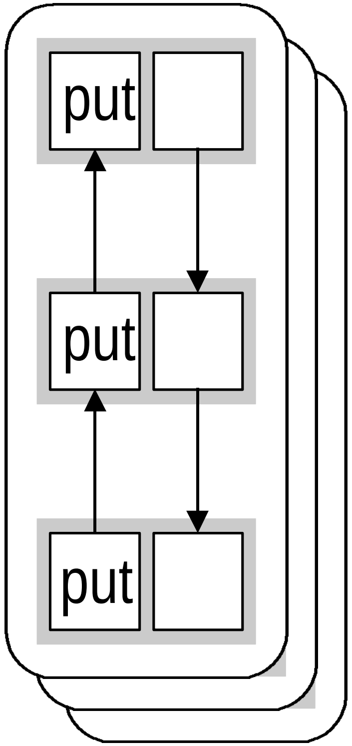
Data Processing

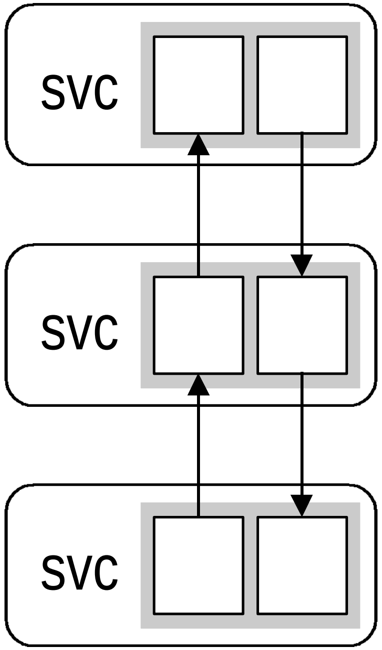
```
int DtProcess::svc(){
  while (1) {
  this->getq(mb);
  if (mb->msg_type() != ACE_Message_Block::MB_HANGUP)
    break;
  if (this->is_reader()) {
    // calc dx/dt, dy/dt, calc strip number, hits, …
    mb = new ACE_Message_Block(fullData);
  } else {
    // calc x,y from chosen strips, duration from energy
    // parameter, …
    mb = new ACE_Message_Block(xytData);
    this->put_next(mb);
  return 0;
}

int DtProcess::put(ACE_Message_Block *m, ACE_Time_Value *t)
{ this->putq(mb); }
```

Variation II of the Theme

- Don’t perform tasks in `svc` routine
  - Perform operations in put routines directly
  - Share the thread of the caller
  - Performance improvement if tasks do only little processing and at high message rates.

Alternative Invocations





High performance
good for single process solutions

Low performance
good for multiple processes

Event Demultiplexing

- A key part in distributed systems
  - Assigning incoming messages to the processors (= ***dispatching***).
  - ***Reacting*** to *timeouts, messages (in/out), interrupts*.
- Is part of other patterns
  - Connection accept, Active object.

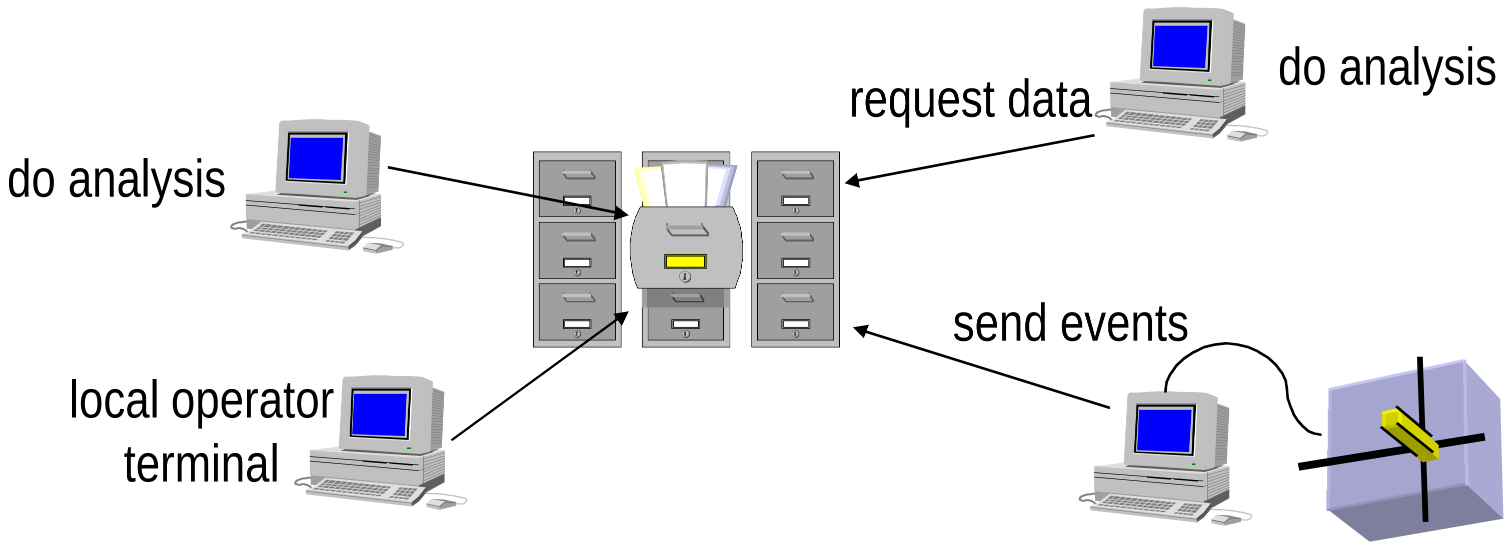
→ ***Reactor and Proactor patterns***

Reactor and Proactor

- ***Reactor***
  - *Handle concurrent (interleaved) service requests*
  - *dispatch requests* to responsible event handlers.
  - ***Synchronous*** event processing
- ***Proactor***
  - ***Demultiplexing to asynchronous operations*** (the process of dispatching is still synchronous),
  - ***Event processing based on completion of events*** (a callback that is also processed synchronously).

Example: Data Server

- Peers want to access files on one machine



First Approach

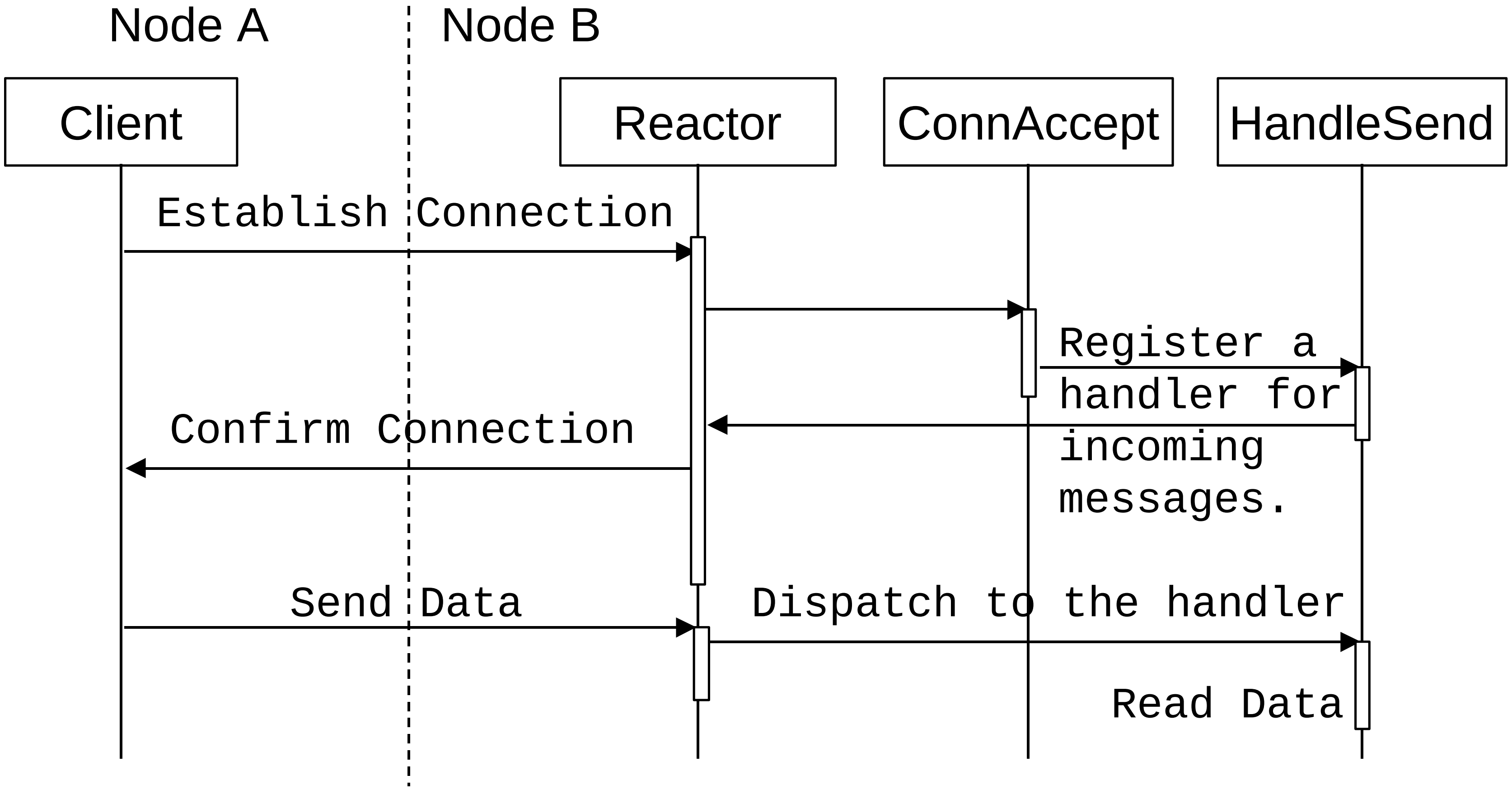
- One thread per connection.
  - Concurrency control will degrade performance.
  - With many threads context switching will influence the quality of the service as well.
  - Portability: Semantics of I/O operations differ on different operating system platforms.
  - Different sources difficult to integrate (stdin, socket).

→ ***Use Reactor pattern***

Reactor

- One handler type for each ***type of service***
  - accept connection, handle input, handle timeout, handle output, close connection.
- ***Register handler*** for an input with Reactor.
- Dispatcher ***synchronously demultiplexes*** and notifies the Reactor object.
- Reactor ***synchronously calls*** back the appropriate event ***handler routine*** that processes the input.

Interaction Diagram



This is a simplified diagram.

Implementation Example

```
class ConnAccept : public Event_Handler {…}
class HandleSend : public Event_Handler {…}

ConnAccept::ConnAccept() {
  Reactor::instance()->register_handler (this, ACCEPT_EVENT);
}

ConnAccept::handle_event() {
  new HandleSend(Handle);
}

HandleSend::HandleSend(HandleT H) {
  Reactor::instance()->register_handler (this, READ_EVENT);
  Reactor::instance()->schedule_timer (this, 0, TIMEOUT);
}

main() {
 Reactor::instance->()run_event_loop(); // Singleton
}
```

Implementation Contd.

```
int HandleSend::handle_input(ACE_HANDLE) {
  // share thread with Reactor
  aHandle.recv(&localBuffer);
  if (localBuffer contains EndOfTransmission marker)
    return -1; // implicitly call handle_close

  … do the database access and send back the values …

  aHandle.send(results);
  return 0;
}

int HandleSend::handle_timeout(ACE_Time_Value& t) {
  return -1;
}

HandleSend::handle_close() {
  aHandle.close(), delete this;
}
```

Service Specification

- ***Single method*** interface
  - `handle_event(`**`EventT`**` Event)` procedure,
  - Switch/case on event in the procedure.
- ***Multiple method*** interface
  - `handle_accept, handle_input, handle_output, handle_timeout, handle_close` procedures.
  - Predefined classes (concrete event handlers) for different events are available in ACE for Acceptor, Connector, Task (Svc_Handler)

Advantages of Reactor

- Separation of concerns
  - ***Dispatching and service*** implementation are ***decoupled*** $\rightarrow$ easier extensibility, reuse services
- Decoupling of application from data transfer
  - Easier design, modification and extension.
- Increased portability.
  - UNIX demultiplexing: `select, poll`
  - WinNT demultiplexing: `WaitforMultipleObjects`
- Serialisation (lock free service implementation).

Disadvantages of Reactor

- Restricted applicability.
  - OS must support abstract handles for all events.
- More difficult to debug than a flat design.
- Non-preemptive
  - Service execution will block further requests.
  - Service routine as threads or Active Objects raises the same problems as in ‘thread per connection’ therefore...

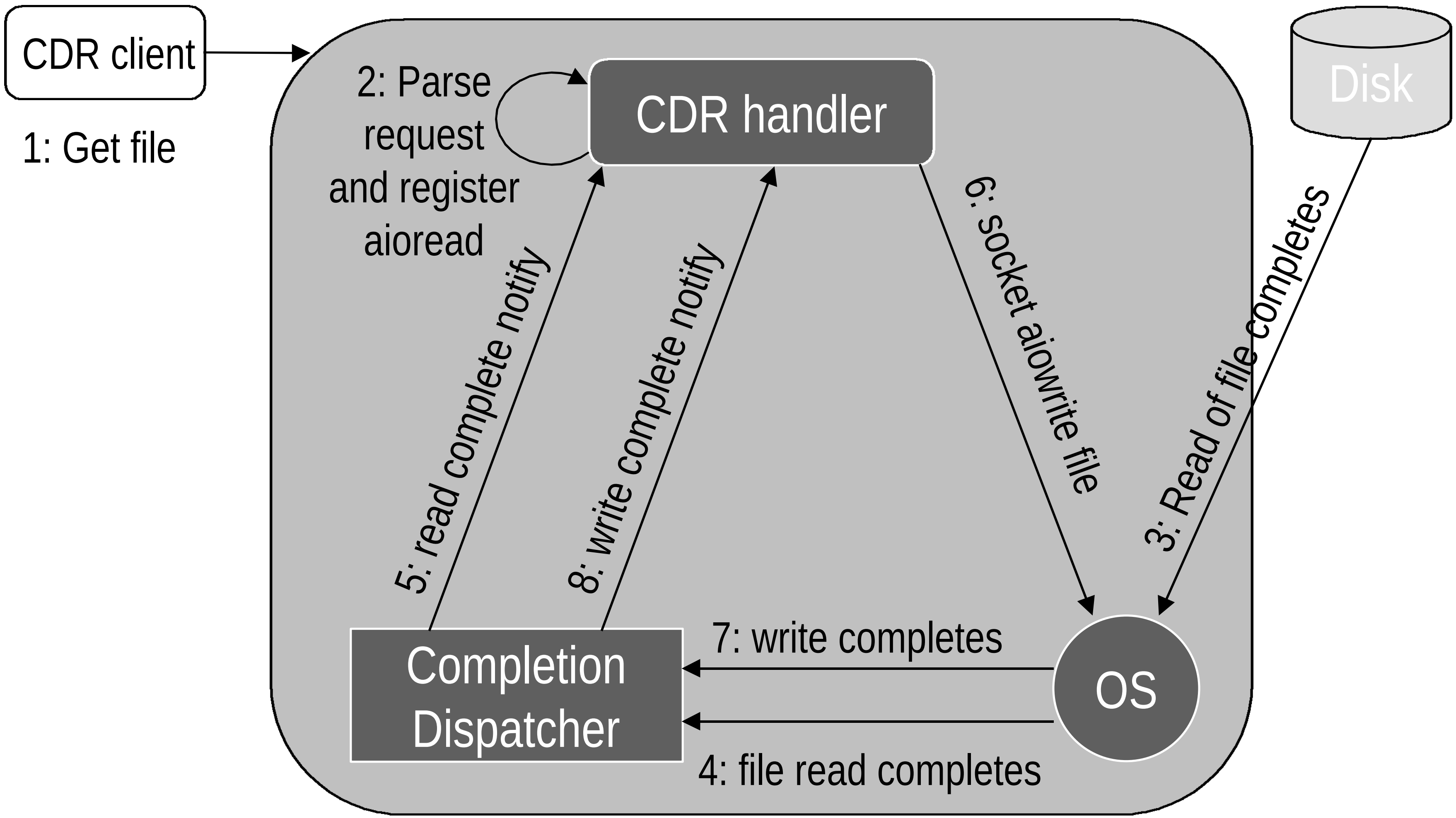
Asynchronous services can use the Proactor

Cut!

Proactor

- For ***handling the completion*** of asynchronous operations.
- The ***result*** of an asynchronous operation ***is queued*** into a well known location.
- A ***callback*** is registered with a ***completion dispatcher*** that notifies the service routine when the operation completes.
- Only applicable if the operating system supports asynchronous operations (`aioread` on Solaris).

Central Data Recording



CDR Request Description

- After a connection a CDR handler is created and socket is read synchronously.
- CDR handler registers and issues an ***asynchronous read*** for the requested data.
- **`aioread`** ***completes*** and the completion and dispatcher notifies the CDR handler.

CDR contd.

- The CDR handler sends the file with a socket **`aiowrite`** command ***asynchronously*** and registers itself as a completion handler.
- After the **`write`** has ***completed*** the OS notifies the completion dispatcher.
- The dispatcher notifies the completion handler.

Implementation Example

```
class CDRHandler : public ACE_Handler {
  // called when read of data from disk completes
  virtual void handle_read_file(
    const ACE_Asynch_Transmit_File::Result& result);
  // called when a write to the socket completes
  virtual void handle_write_stream(…);
  ACE_Handle handle (void) const {
    return this->outputStream.get_handle(); // handle for ws
  }
  ACE_Asynch_Write_Stream ws;  // for writing to socket
  ACE_Asynch_Read_File    rf;  // for reading data from disk
};

CDRHandler::CDRHandler() {
  ws.open(*this); // uses handle from outputStream
  ACE_Message_Block *mb = new ACE_Message_Block(LENGTH);
  rf.open(*this, fhandle); // pass self as completion handler
  rf.read(*mb, mb->size());
}
```

Implementation contd.

```
void CDRHandler::handle_read_file (const
    ACE_Asynch_Read_File::Result& result)
{
  if (result.success()) {
    this->ws.write(result.message_block(),
                   result.bytes_transferred ());
    if ( file size > size transferred )
      … initiate another asynchronous read …
  }
};

void CDRHandler::handle_write_stream( … ) {
 if (result.success()) {
   n = result.bytes_to_write()-result.bytes_transferred();
   if (n != 0)
     … initiate another asynchronous write …
   else
     ws.close, rf.close(), done = 1;
}
```

CDR Main

```
static int done = 0;

int main (int argc, char* argv[]) {
  … accept a connection from a client using reactor …
  CDRHandler handler;
  while (!done)
    ACE_Proactor::instance()->handle_events();

  return 0;
}
```

Benefits and Drawbacks

- It is possible to have more than one requests that work ***interleaved*** without having the complexity of multiple threads/processes.
- ***But*** asynchronous operations may lead to indeterministic behaviour.
- ***Introducing state information*** into the completion handler complicates programming.
  - Imagine the case of several clients

Service Configuration

- Each beam test environment will have its own, *specific "reconstruction" algorithms*
  - Microstrip gas chambers
  - pixel detectors
  - calorimeters (crystal arrays)
- The service modification must be transparent for clients
  - For those who write the interface to the OODBMS
  - For those who write the on-line framework

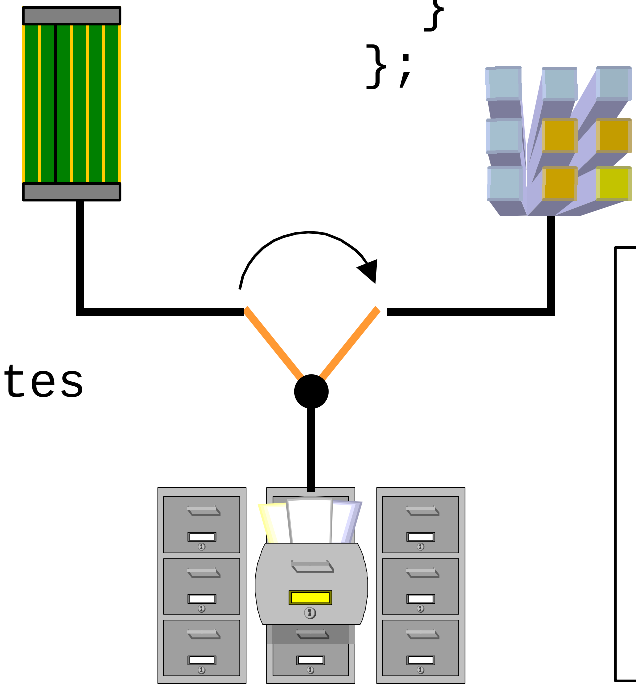
Service Configurator Pattern

- ***Decouples*** the ***behaviour*** of services ***from*** the point in time at which service implementations are ***configure***d.
- ***Use*** it when services shall be initiated, suspended, resumed and terminated dynamically.
  - All other use cases fall back to this one.
- Do ***not use*** in case of ***security*** restrictions or when the service is ***coupled too tightly*** to its context.

Exchange a Service on the Fly

```
class TBAnalysis
   :ACE_Service_Object
{
  handle_input(ACE_Handle fd)
  {
    ReconstructMSGC();
  }


  init (int argc,
           char** argv)
  {
    // get data from
    // different crates

    SetPortNumber;
  }

};

TBAnalysis analysis;
```

```
class TBAnalysis
   :ACE_Service_Object
{
  handle_input(ACE_Handle fd)
  {
    ReconstructECAL;
  }
};
```



```
#svc.conf file:
#Terminate Handler
remove TBAnalysis
#Reconfigure
dynamic TBAnalysis
Service_Object *
~/ECAL.dll:analysis
“-Port 5005”
```

Service Repository

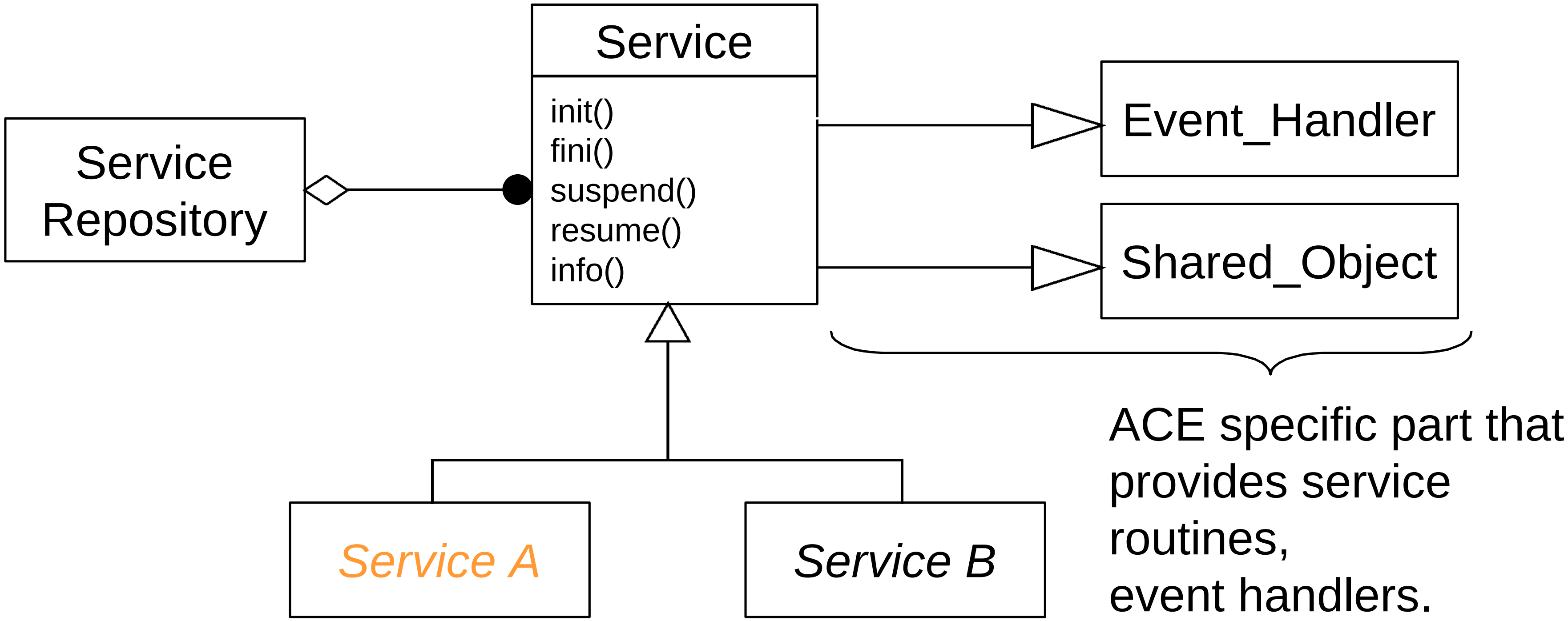
- The Service Repository centrally ***manage***s the configured ***concrete services*** of the application.
- A ***configuration file*** (`svc.conf`) is used to interface to this Service Repository.
- Services can be
  - configured as *static* *or* *dynamic*,
  - *add*ed and *remove*d,
  - *suspend*ed and *resumed*,
  - *Modules* can be pushed onto, popped from *stream*.

The Service

- Is a class that ***offers a predefined interface*** to dynamically configure a service.
  - `init(int argc, char** argv)` is the entry point of the service and called automatically when the service get activated.
  - `fini(void)` serves as a hook for implementing controlled removal of the service.
  - `suspend/resume` may be implemented.
  - `info(char**, size_t)` - implement this to provide information about the service.

Concrete Service

- Inherits from Service and ***contains*** a concrete ***implementation*** of the service.



Where to Use

- Services have to be *configured at run-time*.
  - Use of pipes, streams, sockets, raw devices, ...
- Implementation of services have to be *exchanged transparently*
  - Compare the Java applet/servlet mechanisms
  - Compare Mobile Agent facilities
- Dynamic reconfiguration
  - for *plug & play like operation*.

Important Issues

- Indeterminism and reduced Reliability.
  - An application that works fine with a specific service or configuration may exhibit completely different behaviour with another one.
- Overhead
  - Could be a time problem in mainstream OS's.
  - VxWorks only knows dynamic linking and is a real-time system, so…
- Complexity of service management.

Concurrency Mechanisms

- Managers for Threads and Processes,
- Guards, atomic operations,
- Conditions,
- Synchronization Wrappers for basic locks
  - Mutex, Semaphore,
  - Barrier.
- The Active Object pattern.

Managers

- Components that contain a set of mechanisms to manage groups of threads or processes.
  - `spawn, suspend, resume, wait.`
- Process manager spawn operation copies of the process image and passes options, whereas a thread manager spawn operation starts a given method as a thread.
- `ACE_Process` and `ACE_Thread` classes exist for direct use of processes and threads.

Mutex

- A **mut**ual **ex**clusion lock is a binary semaphore (implementing a spin-lock algorithm)
- for controlling access to ***one*** shared ***resource***.
- Typical interface:
  - `acquire`, `try_acquire`, `release`
- Available for threads and processes.

RW_Mutex

- Readers/writers lock is applicable if a resource is rather read than modified.
- Not available in POSIX or Win32, but ACE implementation is available for all OS's.
- Several tasks may acquire a read lock. Only if the writer is in the critical section they are blocked.
- Getting a write lock is only possible if all read/write locks are free.

Semaphores, Recursive Mutex

- The “usual” Semaphore behaviour…
  - Decrement semaphore on acquire and block if semaphore `value < 0`
  - Become unblocked is semaphore `value = 0`
  - Increment semaphore value on unlock.
- Recursive Mutexes may be reacquired by the same thread/process (e.g. for callbacks where the service routine may be reentered while the other one is waiting for a resource).

Guards

- A more convenient way to use Locks.
- A guard may work with any lock type.
  It is a template class
  - `Lock::acquire` is called in the CTOR of class `Guard`
  - `lock::release` in the DTOR.

```
void critical()
{
   ACE_Guard <ACE_Semapore> guard(GlobalSem);
  … do the critical job …
}
```

Atomic Operations

- Similar to guards, ACE provides a template class for atomic operations.
- Includes the “usual operators” for basic types `++, --, +=, -=, ==, >=, = <=, …`

```
ACE_Atomic_Op <ACE_Thread_Mutex, int> cThreads;

int svc(){
  cThreads++;
  doSomething();
  cThreads--;
  …
}
```

Condition Variables

- The `ACE_Condition<class MUTEX>` class is used to ***block on a change in a state*** of a condition variable.
- The task acquires the mutex and then waits on the condition. If it is false the mutex is unlocked and the task is suspended
  (the mutex is locked for a very short time only).
- The task that wants to signal a condition (one or all waiting threads) also acquires the mutex first.

Example for Condition

- Autocontrol DAQ system
  - tasks on the same machine

```
spawnAnalysis();
analysis.broadcast();
```

```
analysis.wait();
readOutStartUp();
readOut.signal();
// Now data flows!
```

```
analysis.wait();
val=readout.wait(DELAY);
if (val==-1) alarm();
else
  displayStatus();
```

Miscellaneous

- Barrier (`Thread_Barrier, Process_Barrier`)
  - Synchronize threads or processes at one rendez-vous point.
- TSS (Thread Specific Storage)
  - Private data that belongs to the thread is made “logically” global to a program.
  - Better performance due to avoidance of locking.
  - Example: the `errno` variable is always global, but returns the last error number of the thread!

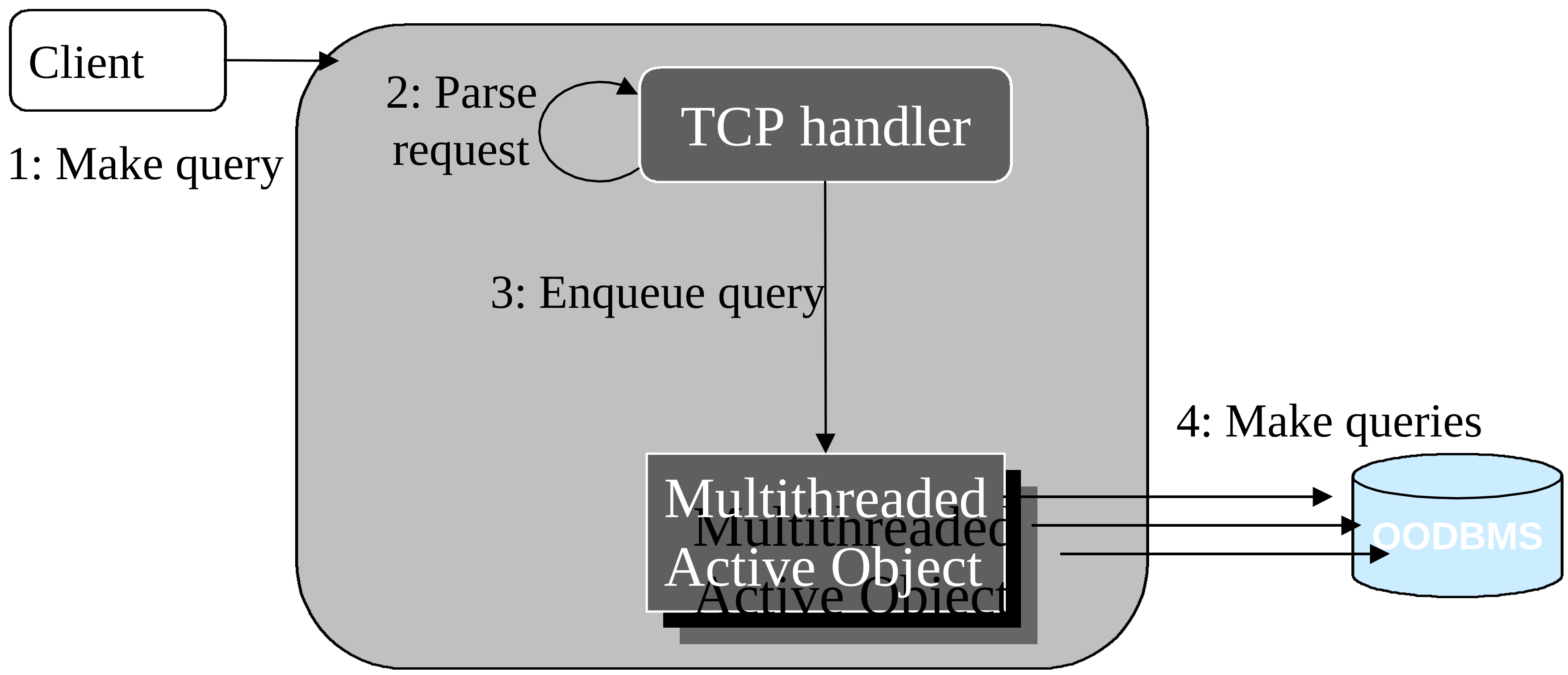
Outlook

- Distributed synchronisation
  - Centralized Token server
  - No transparent distributed locking, no condition variables or barriers yet.
- Deadlock detection algorithm available
  - `check_deadlock(ACE_Token_Proxy *proxy)` returns `0` if `acquire` causes a deadlock.
  - Only for use with the token proxy, not ordinary semaphores, mutexes, etc.

Task

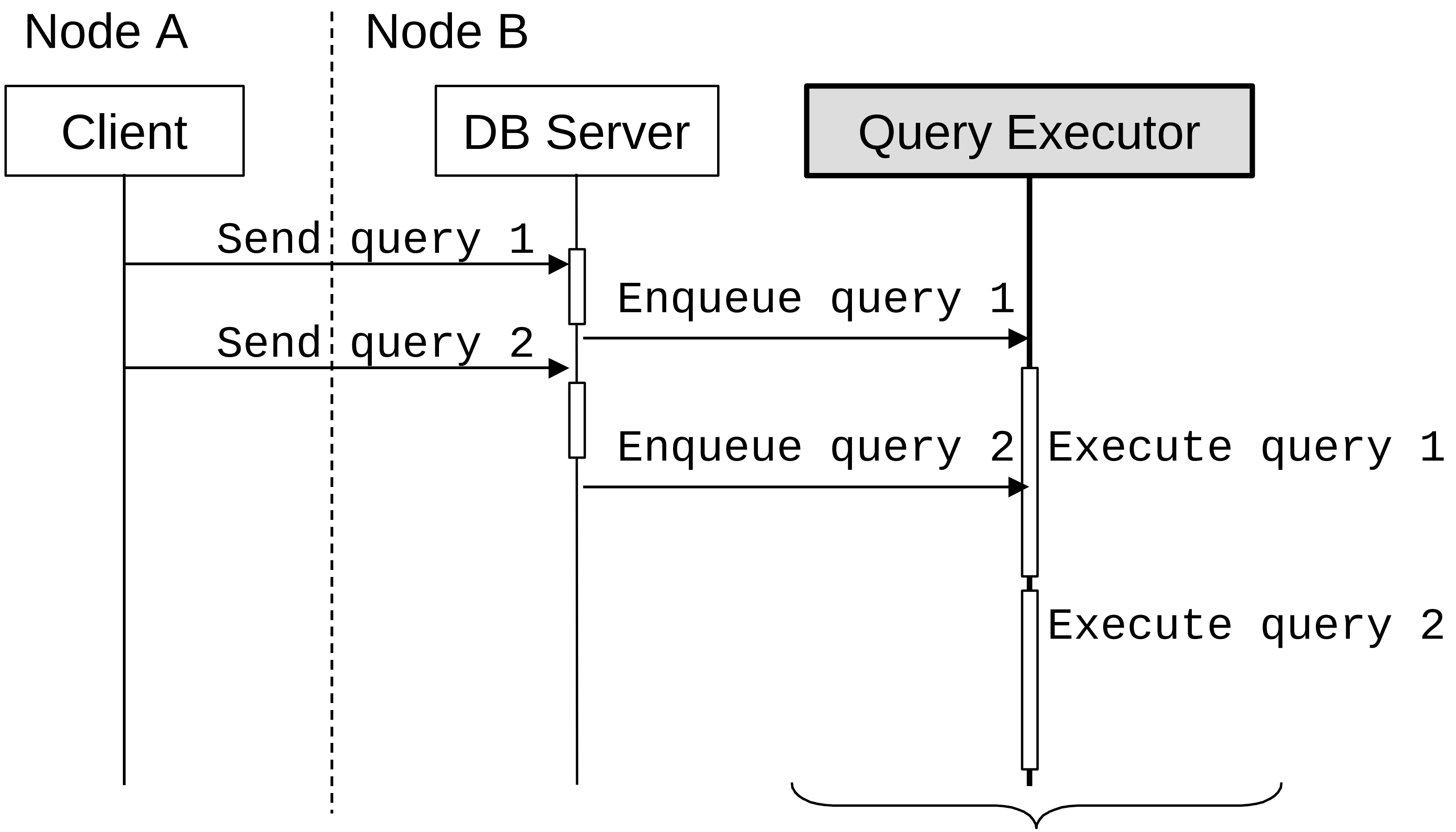
- Method ***execution is decoupled from*** method ***invocation*** in order to simplify synchronized access to a shared resource.
- Can be used as in a stream.
- Suitable for producer/consumer problems.
- For taking advantage of parallelism.
- Alleviates clients from being blocked and simplifies the implementation of servers.
  - Requests can be queued

Data Base Query Server



Sending back a reply is not shown here.

Interaction Diagram



The Active Object may be multithreaded

ACE Task

```
class QExecutor : ACE_Task <ACE_MT_SYNCH>
{
  Qexecutor(int n_threads) {
       // Make use of a multithreaded system
       this->activate(THR_NEW_LWP, n_threads);
  }

  int put (ACE_Message_Block *mb) {
       return this->putq (mb);
  }

  int svc() {
       ACE_Message_Block *mb;
       for (;;) {
              this->getq(mb); // get the query
              SQL_Query (mb->base); // execute it!
       }
};
```

Active Object

- Client calls the objects method, but...
- A ***Method Object*** is queued.
- State information is encapsulated together with the actions that shall be executed in the thread.
- Different behaviours possible (query, update)
- The active object retrieves the method object from a queue and calls a hook.
- The queue for the method objects can be used for ***scheduling*** these objects (RT!).

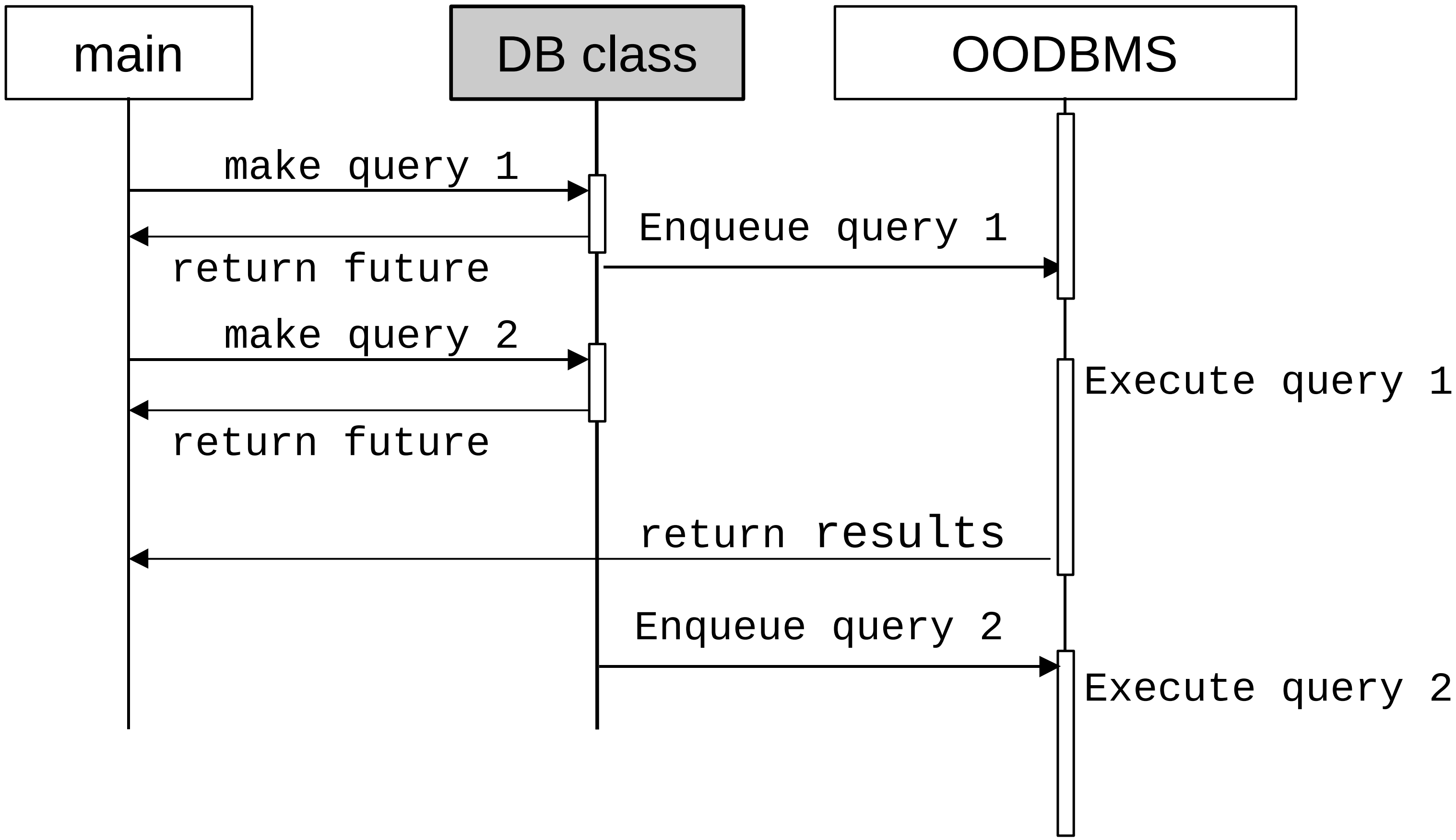
Example - Retrieve Event

```
class DB : ACE_Task <ACE_MT_SYNCH>
{
  int open() { this->activate(THR_NEW_LWP); }

  int svc() {
      ACE_Message_Block *mb;
        auto_ptr<ACE_Method_Object> mo
          (this->activation_queue_.dequeue ());
            mo->call(); // get the query
      }
  ACE_Future<EcalEvent> getEcalEvent([…]) {
    ACE_Future<EcalEvent> result;
    this->activation_queue_.enqueue
      (new EcalMO(this, […], result));
    return result;
  }
};
```

MO returns Future

```
class EcalMO : public ACE_Method_Object {
  EcalMO (DB* db, […], ACE_Future<EcalEvent>& r){
  // make local copies of the parameters }

  virtual int call() {
    // make the query to the OODBMS
    EcalEvent e = oodbms_.retrieve([…]);
    return this->futureResult_.set (e);
  }
};

int main (int argc, char* argv[]) {
  DB db("jun98", "aug98", "H2");
  ACE_Future<EcalEvent> e[100];
  for (int i = 0; i < 99; i++)
    e[i] = db.getEcalEvent([…]); // asynchronous

  for (int i = 0; i < 99; i++)
    e[i].get(tmpEvent), doAnalysis(tmpEvent);
```

Trace with Futures



ACE Size

- Solaris 2.7, without debug information:
  - 2.4 MB as shared library (5.4 MB w. debug info)
  - 3.3 MB as static library
- Build of library components possible
  - `gmake ACE_COMPONENTS=OS …`
  - OS, Utils, Logging, Threads, Demux, Connection, Sockets, IPC, Svcconf, Streams, Memory, Token, Other
  - No consistency checks available, user has to know, what his program needs

ACE Outside CERN

- BaBar (SLAC)
  - Level-3 trigger farm software, distributed histograms
- DØ (Fermilab)
  - Level-3 and VME readout
- High Frequency Active Aurorial Research Program "HAARP" (Air Force and Naval RL)
  - used for control and data acquisition
- Merril Lynch
  - Option trading desk software

More Information

- ACE
  http://www.cs.wustl.edu/~schmidt/ACE.html
- TAO (The ACE ORB)
  http://www.cs.wustl.edu/~schmidt/TAO.html
  - CORBA 2.2 compliant ORB, real-time extensions
- Newsgroup: comp.soft-sys.ace
- Commercial Support
  - www.riverace.com

Johannes.Gutleber@cern.ch